Available online at www.sciencedirect.com
ScienceDirect
Electronic Notes in Theoretical Computer Science
ELSEVIER
Electronic Notes in Theoretical Computer Science 265 (2010) 49–71
www.elsevier.com/locate/entcs

Compositional System Security with Interface-Confined Adversaries

Deepak Garg, Jason Franklin, Dilsun Kaynar and Anupam Datta

Carnegie Mellon University
Pittsburgh PA, USA

Abstract

This paper presents a formal framework for compositional reasoning about secure systems. A key insight is to view a trusted system in terms of the interfaces that the various components expose: larger trusted components are built by combining interface calls in known ways; the adversary is confined to the interfaces it has access to, but may combine interface calls without restriction. Compositional reasoning for such systems is based on an extension of *rely-guarantee* reasoning for system correctness [27,21] to a setting that involves an adversary whose exact program is not known. At a technical level, the paper presents an expressive concurrent programming language with recursive functions for modeling interfaces and a logic of programs in which compositional reasoning principles are formalized and proved sound with respect to trace semantics. The methods are illustrated through a small fragment of an idealized file system.

Keywords: Compositional system security, program logic, temporal logic

1 Introduction

Compositional security is a recognized scientific challenge for trustworthy computing (see, for example, Bellovin [5], Mitchell [28], Wing [35]). Contemporary systems are built up from smaller components. However, even if each component is secure in isolation, the composed system may not achieve the desired end-to-end security property: an adversary may exploit complex interactions between components to compromise security. Such attacks have shown up in the wild in many different settings, including web browsers and infrastructure [3,4,20,10,19], network protocols and infrastructure [26,2,29,22,35], and application and systems software [34,8]. While there has been progress on understanding secure composition in specific settings, such as information flow control for non-interference-style properties [24,23,25] and cryptographic protocols [17,9,31,12,6,11], a systematic understanding of the general problem of secure composition has not emerged yet.

This paper makes a contribution in this space. We present a formal framework for compositional reasoning about secure systems, incorporating two main insights.

doi:10.1016/j.entcs.2010.08.005

First, we posit that a general theory of secure composition should enable one to flexibly model and parametrically reason about different classes of adversaries. This is critical because while specific domains may have a canonical adversary model (e.g., the standard network adversary model for cryptographic protocols), it is unreasonable to expect that a standard adversary model can be developed for all of system security. To develop such a theory we view a trusted system in terms of the interfaces that the various components expose: larger trusted components are built by combining interface calls in known ways; the adversary is confined to the interfaces it has access to, but may combine interface calls without restriction. Such *interface-confined adversaries* are precisely modeled in our framework and provide a generic way to model different classes of adversaries. For example, in virtual machine monitor-based secure systems, we can model an adversarial guest operating system by confining it to the interface exposed by the virtual machine monitor (VMM). Similarly, adversary models for web browsers, such as the *gadget adversary* [4], can be modeled by confining the adversary to the read and write interfaces for frames guarded by the same origin policy as well as the frame navigation policies.

Second, we develop compositional reasoning principles for such systems in a rich logic of programs. A salient feature of our logic is that program specifications can use temporal operators, which allows them to relate not only the states and actions at the beginning and end of a program, but also at all points in between. This is essential because most properties of interest to us are safety properties [1] that must hold throughout a program's execution. In this regard our work is similar to prior work on analysis of network protocols [13]. We further enrich the reasoning principles by extending ideas from *rely-guarantee* reasoning [27,21]. While rely-guarantee reasoning was developed for proving correctness properties of known concurrent programs, we extend it to soundly reason about system security in the presence of interface-confined adversaries. These principles generalize prior work on compositional logics for network protocol analysis [12,13,33,18] and secure systems analysis [14] and are also related to a recently proposed type system for modularly checking interfaces of security protocols [6] (see Section 2 for a detailed comparison).

At a technical level, the paper presents an expressive concurrent programming language with recursive functions for modeling system interfaces and interface-confined adversaries. Specifically, the programming language is based on an untyped, first-order, concurrent version of the lambda-calculus with side-effects (Section 3 presents more details). Security properties are specified in a logic of programs (also described in Section 3). Compositional reasoning principles are codified in the proof system for the logic of programs to support modular reasoning about program specifications (Section 4.1), trusted programs whose programs are known (Section 4.2) and interface-confined adversarial (untrusted) code (Section 4.3). We present the formal semantics for the logic of programs and the main technical result of the paper—a proof of the soundness of the proof system with respect to the trace semantics of the logic (Section 5). Finally, we describe how the proof rules support rely-guarantee reasoning in the presence of adversaries (Section 6). Concluding remarks and directions for future work appear in Section 7.

We illustrate our methods by applying them to an idealized file system whose access control matrix protects its own integrity through a special 'administrate' permission, and a homework administration application running on the file system. The interface-based view is useful both in modeling the file system's interfaces that are composed in known ways by the application and in flexibly modeling the adversary. Our illustrative proofs exercise all compositional reasoning principles developed in this paper. Further examples, including a simplified web mashup and an analysis of secrecy in the Kerberos V5 protocol can be found in the full version of this paper [16].

2 Related Work

We discuss below closely related work on logic-based and language-based approaches for compositional security. Orthogonal approaches to secure composition of cryptographic protocols include work on identifying syntactic conditions that are sufficient to guarantee safe composition [17,11]. Another orthogonal approach to secure composition is taken in the *universal composability* or *reactive simulatibility* [9,31] projects. These simulation-based definitions when satisfied can provide strong composition guarantees. However, they have been so far applied primarily to cryptographic primitives and protocols.

Compositional Logics of Security. The framework presented in this paper is inspired by and generalizes prior work on logics of programs for network protocol analysis [12,13,33,18] and secure systems analysis [14]. At a conceptual level, a significant new idea is the use of interface-level abstractions to modularly build trusted systems and flexibly model adversaries with different capabilities by confining them to stipulated interfaces. In contrast, prior work lacked the interface abstraction and had a fixed adversary. Also, the actions (side-effects) in the language were fixed in prior work to communication actions, cryptographic operations, and certain operations on shared memory. On the other hand, our programming model and logic are parametric in actions. One advantage of this generality is that the compositional reasoning principles (proof rules) are action-independent and can be applied to a variety of systems, thus getting at the heart of the problem of compositional security (see Section 3.1 for details of the parametrization). We expect domain-specific reasoning to be codified using axioms; thus, the set of axioms for reasoning about network protocols that use cryptographic primitives will be different from those for reasoning about trusted computing platforms. The treatment of rely-guarantee reasoning in the presence of adversaries generalizes the invariant rule schemas for authentication [12], integrity [14], and secrecy [33] properties developed earlier.

Refinement types for verifying protocols. Recently, Bhargavan et al. have developed a type system to modularly check interfaces of security protocols, implemented it, and applied it to analysis of secrecy properties of cryptographic protocols [6]. Their approach is based on refinement types, i.e, ordinary types qualified

with logical assertions, which can be used to specify program invariants and pre- and post-conditions. Programmers annotate various points in the model with assumed and asserted facts. The main safety theorem states that all programmer defined assertions are implied by programmer assumed facts in a well-typed program. However, a semantic connection between the program state and the logical formulas representing assumed and asserted facts is missing. In contrast, we prove that the inference system of our logic of programs is sound with respect to trace semantics of the programming language. Our logic of programs may provide a semantic foundation for the work of Bhargavan et al. and, dually, the implementation in that work may provide a basis for mechanizing the formal system presented in this paper. Bhargavan et al.'s programming model is more expressive than ours because it allows higher-order functions. We intend to add higher-order functions to our framework in the near future.

3 Programming Model and Security Properties

We start by describing our formalism for modeling systems and a logic of programs for specifying and reasoning about their security properties. In Section 3.1, we describe a concurrent programming language for modeling systems. Section 3.2 introduces our running example. Section 3.3 presents the logic of programs that is used to express security properties of systems modeled in the programming language.

3.1 Programming Model

We model a system as a set of concurrent threads, each of which executes a sequential program. The threads may or may not be located on the same machine. The program of each thread may either be available for analysis, in which case we call the thread trusted, or it may be unknown, in which case we call the thread untrusted or adversarial. The program of a thread consists of atomic steps called *actions* and control constructs like conditionals and sequencing. Actions model all operations other than control flow including side-effect free operations like encryption, decryption, and cryptographic signature creation and verification, as well as inter-thread interaction through network message sending and receiving, shared-memory reading and writing, etc. In the formal description of our programming model, its operational semantics, and reasoning principles, we treat actions abstractly, denoting them with the letter a in the syntax of programs and representing their behavior with sound axioms in the logic of programs. The soundness theorem presented in this paper is general, and applies whenever axioms chosen to codify properties of actions are sound.

In the interest of security, programs may not have access to all actions used to model a system. Instead, programs may be limited to using a set of trusted interfaces that are exposed by the system. Formally, an interface is a function $f(x) \stackrel{\triangle}{=} e$, with name f, argument x, and body e. The body may execute actions and interfaces, including itself, thus allowing for both recursion and mutual recursion. Recursive

interfaces are important in many settings especially servers that listen for client requests, process them, and then loop back to the listening state.

Formally, the sequential program of each thread is described by an expression e in the following language. t denotes a *term* that can be passed as arguments, and over which variables x range. We do not stipulate a fixed syntax for terms; they may include integers, Booleans, keys, signatures, tuples, etc. However, our language is first-order, so terms may not contain expressions. To simplify reasoning principles, the language is interpreted functionally: all expressions and actions must return a term to the calling site and variables bind to terms. Mutable state, if needed, may be modeled as a separate syntactic entity whose contents can be updated through actions, as illustrated in an example later in this section.

$$\begin{array}{ll} \text{Expressions} & e ::= t \mid \texttt{act}\ a \mid \texttt{let}(e_1, x.e_2) \mid \texttt{if}(b, e_1, e_2) \mid \texttt{call}(f, t) \\ \text{Function defns} & ::= f(x) \stackrel{\triangle}{=} e \end{array}$$

The expression t returns term t to its caller. $\texttt{act}\ a$ evaluates the action a, potentially causing side-effects. $\texttt{let}(e_1, x.e_2)$ executes e_1 first, then binds the term obtained from its evaluation to the variable x and evaluates e_2. $\texttt{if}(b, e_1, e_2)$ evaluates e_1 if b is true and evaluates e_2 otherwise. $\texttt{call}(f, t)$ calls function f with argument t: if $f(x) \stackrel{\triangle}{=} e$ then $\texttt{call}(f, t)$ evaluates to $e\{t/x\}$. ($\Xi\{t/x\}$ denotes the usual capture-avoiding substitution of the term t for the variable x in the syntactic entity Ξ.)

Operational Semantics. The operational semantics of our programming language define how a configuration, the collection of all simultaneously executing threads of the system and shared state, reduces one step at a time. Formally, a configuration $\mathcal{C}$ contains a set of threads $T_1, \ldots, T_n$ and a shared state σ. We treat the state abstractly in our formal specification; it may be instantiated to model shared memory as well as the network which holds messages in transit depending on the application. The state may change when threads perform actions, e.g., a send action by one thread may add an undelivered message to the network part of the state. Such interaction between the state and actions is captured in the reduction rule for actions, as described below.

A thread is a triple $I; K; e$ containing a unique thread identifier I, an execution stack K and an expression e that is currently executing in the thread (also called *active* expression of the thread) . The execution stack records the calling context of the active expression as a sequence of frames.

$$\begin{array}{ll} \text{Thread id} & I \\ \text{Frame} & F ::= x.e \\ \text{Stack} & K ::= [] \mid F :: K \\ \text{Thread} & T ::= I\,;K\,;e \\ \text{Configuration} & \mathcal{C} ::= \sigma \rhd T_1, \ldots, T_n \end{array}$$

$$\frac{\sigma\,;I \rhd a \mapsto \sigma'\,;I \rhd t \qquad \texttt{eval}\ t\ t'}{\sigma \rhd I\,;(x.e) :: K\,;\texttt{act}\ a \hookrightarrow \sigma' \rhd I\,;K\,;e\{t'/x\}}\text{red-act}$$

$$\frac{}{\sigma \rhd I\,;K\,;\texttt{let}(e_1, x.e_2) \hookrightarrow \sigma \rhd I\,;(x.e_2) :: K\,;e_1}\text{red-let}$$

$$x\frac{\sigma \rhd T_i \hookrightarrow \sigma' \rhd T_i'}{\sigma \rhd T_1, \ldots, T_i, \ldots, T_n \longrightarrow \sigma' \rhd T_1, \ldots, T_i', \ldots, T_n}\text{red-config}$$

Fig. 1. Operational semantics, selected rules

Selected reduction rules for our language are shown in Figure 1. (Remaining rules are in the full version of this paper [16].) The rules are parametric in judgments for evaluating actions a and terms t, both of which are treated abstractly. The judgment $\texttt{eval}\ t\ t'$ means that term t evaluates completely to the term t'. For example, in a model that includes standard arithmetic, a concrete instance of the judgment would be $\texttt{eval}\ (3+5)\ 8$. The judgment $\sigma\,;I \rhd a \mapsto \sigma'\,;I \rhd t$ means that action a when executed in thread I updates the shared state σ to σ' and returns term t. As an example, in a system with shared memory, σ may be a map from locations l to terms t, the action $\texttt{write}\ l, t$ may change contents of location l to t, and the following judgment may hold: $\sigma\,;I \rhd \texttt{write}\ l, t \mapsto \sigma[l \mapsto t]\,;I \rhd ()$.

Our primary reduction relation, that for configurations, has the form $\mathcal{C} \longrightarrow \mathcal{C}'$. It interleaves reductions of individual threads in the configuration in an arbitrary order (rule red-config). Reductions of threads are formalized by a reduction relation for threads: $\sigma \rhd T \hookrightarrow \sigma' \rhd T'$. This reduction relation is standard for functional programs with side-effects and we omit its description. Reductions caused by execution of actions are called *effectual reductions* (rule red-act in the figure) whereas others are called *administrative reductions*.

Our reasoning principles establish properties of *traces*. A trace $\mathcal{T}$ is a finite sequence of reductions $\mathcal{C} \longrightarrow \mathcal{C}'$ starting from an initial configuration. We associate a time point u with every reduction on a trace. A time point is either a real number or $-\infty$ or ∞. The only constraint on time points associated with a trace is that they be monotonically increasing along the trace. Diagrammatically, we represent a trace as follows (u_i is the time at which $\mathcal{C}_{i-1}$ reduces to $\mathcal{C}_i$.)

$$\xrightarrow{u_0} \mathcal{C}_0 \xrightarrow{u_1} \mathcal{C}_1 \ldots \xrightarrow{u_n} \mathcal{C}_n$$

We assume that individual reduction steps happen instantaneously at the time associated with them on the trace and that the state σ on the trace does not change between two consecutive reductions. Specifically, in the above illustration, if $\mathcal{C}_i = \sigma_i \rhd T_{i,1}, \ldots, T_{i,n}$, then we assume that the state of the trace is σ_i in the interval $[u_i, u_{i+1})$. u_0 is called the *start time* of the trace. In addition to the sequence of reductions and time points, a trace may also include auxiliary information, such as contents of state at the start time.

File Mutable Data Structures

C : filename $\rightarrow$ data
P : (filename $*$ prin $*$ perm) set

File System Interfaces

$\mathcal{F} = \{\texttt{permadd}, \texttt{readfs}, \texttt{writefs}, \texttt{send}, \texttt{recv}, \texttt{lock}, \texttt{unlock}\}$

```
permadd(f, k, p) ≜
  lock(P);
  let s = read(P) in
  if ((f, self_prin, ad) ∈ s)
  then unlock(P); insert(P, (f, k, p))
  else unlock(P)

readfs(f) ≜
  lock(P);
  let s = read(P) in
  if ((f, self_prin, re) ∈ s)
  then unlock(P); C(f)
  else unlock(P)

writefs(f, d) ≜
  lock(P);
  let s = read(P) in
  if ((f, self_prin, wr) ∈ s)
  then unlock(P); update(C, f, d)
  else unlock(P)
```

Files and Principal in Classroom

Special principal: Teacher
N students: $s[1], \ldots, s[N]$
N homework files: $f[1], \ldots, f[N]$

Initial Access Control Matrix, P

$\{(f[1], \text{Teacher}, \text{ad}), \ldots, (f[N], \text{Teacher}, \text{ad})\}$

Known Programs

Code of principal Teacher:

```
teacher_body = teacher_loop();

teacher_loop() ≜
  (n, p) = recv;
  if (p = re ∨ p = wr)
  then permadd(f[n], s[n], p)
  else ();
  teacher_loop()
```

Fig. 2. File system with access control (left) and an application on it (right)

3.2 A File System Example

We introduce an illustrative, running example of an idealized file system that uses an access control matrix for security. We represent the file system using two abstract, mutable data structures that are part of the shared state σ from Section 3.1: a map C from file names to file contents, and a set P of tuples (file name, principal, permission), which represents the access control matrix. Principal k has permission p on file f if and only if $(f, k, p) \in P$. We use three permissions: read (re), write (wr), and administrate (ad). The last permission allows a principal to add permissions for other principals on the relevant file. Both data structures are shared between the file system and users, some of whom may be malicious.

Actions. Our model includes several primitive actions a, which we describe informally and whose formal details we elide. Two actions – $C(f)$ and $\texttt{update}(C, f, d)$ – read the file f and write the contents d to file f, respectively (recall that C is the map from file names to their contents). Note that these actions do not make any access checks; the latter are performed by interfaces that wrap the actions as described below. Actions $\texttt{lock}(P)$ and $\texttt{unlock}(P)$ obtain and release an exclusive-

write lock on the access control matrix P. The lock is necessary to avoid write-write and read-write races on the matrix. Action $\texttt{read}(P)$ reads the access control matrix, while $\texttt{insert}(P, (f, k, p))$ inserts the triple (f, k, p) into the access control matrix. Finally, we have actions $\texttt{send}(m)$, that sends a message m on the network (we assume that the source and destination fields of a message can be spoofed, so we do not model them), and `recv` that receives a message from the network.

Interfaces. Since the actions $C(f)$, $\texttt{update}(C, f, d)$, and $\texttt{insert}(P, (f, k, p))$ do not perform any access control checks, they are not directly accessible to programs. Instead, programs are allowed to execute these actions only through fixed interfaces that the file system exposes. This is characteristic of our modeling approach: we limit programs to stipulated interfaces that make security relevant checks, even though low-level actions may not be secure in themselves. Interfaces relevant to this example – `readfs`, `writefs`, and `permadd` – are shown in Figure 2. For better readability, we omit the keywords `act` and `call` from programs and write $\texttt{let}(e_1, x.e_2)$ as $\texttt{let}\ x = e_1\ \texttt{in}\ e_2$. We assume that each thread runs on behalf of a principal, whose permissions apply to the thread. The term `self_prin` in a program dynamically binds to the principal who owns the executing thread (*a la* the system call `getuid`() in POSIX). The body of each interface checks relevant access permissions in P before calling $C(f)$, $\texttt{update}(C, f, d)$, or $\texttt{insert}(P, (f, k, p))$. Programs may use the other actions `send`, `recv`, `lock`, and `unlock` directly, so the set of primitives available to programs is $\mathcal{F} = \{\texttt{permadd}, \texttt{readfs}, \texttt{writefs}, \texttt{send}, \texttt{recv}, \texttt{lock}, \texttt{unlock}\}$.

Application Program. As an application of our file system, we consider a homework administration system. Assume that a classroom containing one principal Teacher and N students $s[1], \ldots, s[N]$ shares our file system. Each student $s[n]$ has a dedicated file $f[n]$ that she can use to submit her homework. The application consists of one server program called `teacher_loop` in Figure 2 run by the teacher and any number of programs run by students (and other adversaries) that are only constrained in that they are confined to the interfaces $\mathcal{F}$ defined above. Initially, the teacher has administrate permission (`ad`) on all files and there are no other permissions. The teacher's program listens to student requests to give them read and write permissions. The program receives a student id n and a permission p over the network, and if the latter is read or write, gives student $s[n]$ permission p over file $f[n]$. (Having so obtained permissions, students can call the interfaces `readfs` and `writefs` to read and write their homework files, respectively.)

Security Property. We are interested in proving the following security property of our application: if file $f[n]$ is updated by thread i, then the owner of thread i is $s[n]$, i.e., only a student can change her homework file. Even though this property may seem obvious, its proof depends on a number of initial assumptions, and properties of interfaces and the program of the teacher, and illustrates all reasoning principles of our framework. The following is an outline of the proof. First, we prove two invariants: 1) The teacher never adds any administrate permission, and 2) In order

to add a permission on a file, a principal must have administrate permission on the file. Together with the assumption that initially only the teacher has administrate permissions, (1) and (2) imply that: 3) Only the teacher ever has administrate permissions. Next, we prove that: 4) The teacher adds write permission on $f[n]$ only for $s[n]$. Finally, we prove that: 5) In order to update a file, a thread must have write permission on the file. Together, (2)–(5) imply our security property.

Technically, properties (1) and (4) follow from an analysis of the *known program* of the teacher, using principles introduced in Section 4.2. Properties (2) and (5) depend on the behavior of all threads, including those that are adversarial, so they must be proved by an *analysis of available interfaces* $\mathcal{F}$ only (Section 4.3). Property (3) requires *rely-guarantee reasoning* because its proof is inductive (Section 6). We sketch some of these proofs formally in subsequent sections.

3.3 Security Properties

We represent security properties as formulas of a logic whose syntax is shown below.

$$\text{Formulas} \quad \varphi, \psi ::= p \mathbin{@} u \mid b \mid t = t' \mid u_1 \leq u_2 \mid \varphi \wedge \psi \mid \varphi \vee \psi \mid \neg\varphi \mid \top \mid \bot \mid \forall x.\varphi \mid \exists x.\varphi$$

The formula $p \mathbin{@} u$ means that the atomic formula p holds at time u (on the trace against which the formula is being interpreted). $p \mathbin{@} u$ is a hybrid modality [32,7]. It is known that all operators of linear temporal logic (LTL) can be expressed using $p \mathbin{@} u$ and quantifiers, so this logic is at least as expressive as LTL. In addition to increasing expressiveness, using a hybrid logic instead of LTL allows us to express order between actions in security properties in an intuitive manner, facilitates reasoning about invariants of programs modularly (Section 4) and also facilitates rely-guarantee reasoning without the need for additional induction principles (Section 6).

b denotes a Boolean term from the syntax of the language. Atomic formulas p are terms applied to predicates, which may represent either the execution of certain actions (corresponding to the language's actions a) or properties of state. For example, a predicate $\texttt{isACM}(s)$ that checks that the access control matrix P is the set s may be defined by saying that $\mathcal{T} \models \texttt{isACM}(s) \mathbin{@} u$ if and only if, in trace $\mathcal{T}$, at time u, the access control matrix is s. Similarly, in protocol analysis, $\mathcal{T} \models \texttt{Enc}(I, t, k) \mathbin{@} u$ may hold if and only if thread I in trace $\mathcal{T}$ encrypts term t with key k at time u.

Example 1. As an illustration, we formalize in our logic the security property described at the end of Section 3.2. Suppose that the predicate $\texttt{Update}(i, f, d)$ holds at time u if at time u, thread i executes the action $\texttt{update}(C, f, d)$ and predicate $\texttt{Owner}(i, k)$ means that principal k owns threads i.[1] Then, the following formula

[1] Technically, the syntax of our logic requires us to write the suffix $\ldots \mathbin{@} u$ after each atomic formula. However, $\texttt{Owner}(i, k) \mathbin{@} u$ is independent of u, so we elide the suffix $\ldots \mathbin{@} u$ after $\texttt{Owner}(i, k)$.

means that only student $s[n]$ can update file $f[n]$.

$$\forall i, u, n, d.\ (\texttt{Update}(i, f[n], d)\ @\ u) \supset \texttt{Owner}(i, s[n])$$

Logic of Programs. On top of the temporal logic described above, we build a logic of programs to reason about properties of traces obtained by executing known or interface-confined programs. Prior experience with security analysis of protocols [12,13,33,18] and trusted computing platforms [14] shows that in addition to standard post-conditions that hold when a program completes execution, analysis of secure systems often requires reasoning about properties that hold while a program executes. We call such properties *invariants* and introduce a novel construct to represent them.

Specifically, we express pre- and post-conditions as well as invariants using six kinds of assertions, generically denoted μ, ν.

$$\begin{aligned}\text{Assertions}\quad \mu, \nu ::= {} & [e]\langle u_b, u_e, i, x\rangle\varphi \mid \{e\}\langle u_b, u_e, i\rangle\varphi \mid \\ & [f]\langle y, u_b, u_e, i, x\rangle\varphi \mid \{f\}\langle y, u_b, u_e, i\rangle\varphi \mid \\ & []\langle u_b, u_e, i\rangle\varphi \mid [a]\langle u_b, u_e, i, x\rangle\varphi\end{aligned}$$

In these assertions, u_b, u_e, i, x, y are bound variables whose scope is φ. (Recall that φ denotes a formula in the temporal logic described earlier.) The intuitive meanings of the six assertions are listed below; formal semantics are postponed to Section 5.

- $[e]\langle u_b, u_e, i, x\rangle\varphi$: If program expression e executes completely during the interval $(u_b, u_e]$ in thread i and returns value x, then φ holds.
- $\{e\}\langle u_b, u_e, i\rangle\varphi$: If program expression e is active in thread i at time u_b and does not return until time u_e, then φ holds.
- $[f]\langle y, u_b, u_e, i, x\rangle\varphi$: If function f is called with argument y in thread i at time u_b and executes completely during the interval $(u_b, u_e]$ returning value x, then φ holds.
- $\{f\}\langle y, u_b, u_e, i\rangle\varphi$: If function f is called with argument y in thread i at time u_b and has not returned until time u_e, then φ holds.
- $[]\langle u_b, u_e, i\rangle\varphi$: If thread i does not perform any effectual reduction in the interval $(u_b, u_e]$, then φ holds. (Recall from Section 3.1 that an effectual reduction is the reduction of an action, as opposed to the reduction of a control flow construct.)
- $[a]\langle u_b, u_e, i, x\rangle\varphi$: If in thread i, the active expression at time u_b is `act` a, and this expression executes at time u_e returning x, then φ holds.

Our reasoning principles (Section 4) are parametric in the variables y,u_b,u_e, i,x and our logic's formal semantics relate assertions to traces for all ground instances of these variables. Assertions $[e]\langle u_b, u_e, i, x\rangle\varphi$ and $[f]\langle y, u_b, u_e, i, x\rangle\varphi$ specify the behavior of programs that complete execution. They are generalizations of partial correctness assertions from other program logic based type-theories like Hoare Type Theory (HTT) [30]. We do not need to specify pre- and post-conditions separately

because we can encode them in φ using the construct p @ u. For example, consider the function $f(x) \triangleq \texttt{let}((\texttt{act read}\ l), z.\ (\texttt{act write}\ l, z + x))$ that increments the contents of the private memory location l by its argument x. We can specify this function in our logic of programs as $[f]\langle y, u_b, u_e, i, x\rangle\ \forall z.\ (\texttt{Mem}(l, z)\ @\ u_b \supset \texttt{Mem}(l, y + z)\ @\ u_e) \wedge x = ()$, where $\texttt{Mem}(l, z)$ means that location l contains value z. Note that the variable x in the body of the function differs from the x in its specification. In the former case, the variable is the argument of the function whereas in the latter case it is the result of the function.

The assertions $\{e\}\langle u_b, u_e, i\rangle\varphi$ and $\{f\}\langle y, u_b, u_e, i\rangle\varphi$ specify invariants of programs – φ holds *while* the program (e or f) is executing. Prior work on Protocol Composition Logic [12,13,33,18] and the Logic of Secure Systems [14] encodes invariants using standard partial correctness assertions and a notion of prefixes of a program. However, prefixes are impossible to define in the presence of function calls and recursion. Our treatment is novel and strictly more general because it allows for specification of invariants of programs with all control flow constructs.

Whereas Section 4 presents a proof system for establishing the four assertions $[e]\langle u_b, u_e, i, x\rangle\varphi$, $\{e\}\langle u_b, u_e, i\rangle\varphi$, $[f]\langle y, u_b, u_e, i, x\rangle\varphi$, and $\{f\}\langle y, u_b, u_e, i\rangle\varphi$, we do not stipulate rules for establishing the remaining two assertions, $[]\langle u_b, u_e, i\rangle\varphi$ and $[a]\langle u_b, u_e, i, x\rangle\varphi$ that specify properties of administrative reductions and effectual reductions, respectively. Instead, these two assertions must be established through sound axioms that would depend on the representation of state and actions chosen; the proof rules for establishing the other four assertions use these two assertions as black-boxes. Our proof system's soundness theorem applies whenever the axioms chosen for establishing these two classes of assertions are sound.

Thread-Local Reasoning. A salient feature of our logic of programs is that the specifications of a program, if established, hold irrespective of actions of threads other than the one in which the program executes. This is implicit in the intuitive meanings of the assertions above as well as the formal semantics of the logic (Section 5). For example, the meaning of $[e]\langle u_b, u_e, i, x\rangle\varphi$ is that if program expression e executes completely during the interval $(u_b, u_e]$ in thread i and returns value x, then φ holds. Since this interpretation does not constrain what other threads do, φ holds *irrespective of the reductions that other threads may have performed in the interim*. As in prior work [13], this local property of the proof system simplifies reasoning significantly since we do not have to reason about reductions of other threads when we wish to prove a property that is specific to a thread (e.g., that the thread does not write a certain location).

Example 2. Suppose that the predicate $\texttt{Insert}(i, (f, k, p))$ holds at time u if thread i executes the action $\texttt{insert}(P, (f, k, p))$ at time u. Then, property (1) from the end of Section 3.2, which states the teacher's program never adds any administrative

permissions, can be expressed as the following invariant of `teacher_body`:

$$\{\texttt{teacher\_body}\}\langle u_b, u_e, i\rangle$$
$$\forall u, f, k.\ ((u_b < u \leq u_e) \supset (\neg \texttt{Insert}(i, (f, k, \texttt{ad})) \ @\ u))$$

4 Compositional Reasoning Principles

Next, we present a proof system that codifies compositional reasoning principles for establishing assertions about programs as well as security properties. In addition to standard rules for proving temporal formulas and syntax-directed rules for proving assertions about programs and functions, our proof system includes two rules of inference that allow deduction of properties of threads from invariants of their programs. We call a thread *trusted* if the program it executes is known, else we call the thread *untrusted* or *adversarial*. Using the first rule (Section 4.2), we may combine knowledge that a particular thread is executing a known program with any invariant of the program to deduce that the formula in the invariant holds forever. The second rule (Section 4.3) embodies our central idea of reasoning about unknown, potentially adversarial, code by confining it to interfaces: if we know that an adversarial thread has access only to certain interfaces, then under certain conditions, we can show that a common invariant of all those interfaces always holds in the system, regardless of the manner in which the adversarial thread uses those interfaces.

Formally, proofs establish one of two hypothetical judgments: $\Sigma; \Gamma \vdash \varphi$ and $\Sigma; \Gamma; \Delta \vdash \mu$. In both judgments Σ is a set of first-order variables that may occur free in the rest of the judgment, Γ is a list of assumed formulas of the temporal logic and Δ contains assumed specifications of functions. A proof establishing either $\Sigma; \Gamma \vdash \varphi$ or $\Sigma; \Gamma; \Delta \vdash \mu$ is parametric in all variables in Σ, i.e., it holds for all ground instances of the variables.

$$\Sigma ::= \cdot \mid \Sigma, x$$
$$\Gamma ::= \cdot \mid \Gamma, \varphi$$
$$\Delta ::= \cdot \mid \Delta, [f]\langle y, u_b, u_e, i, x\rangle\varphi \mid \Delta, \{f\}\langle y, u_b, u_e, i\rangle\varphi$$

The judgment $\Sigma; \Gamma \vdash \varphi$ coincides with the standard hypothetical judgment of first-order classical logic with equality (we treat $p \ @\ u$ as an atomic formula $p(u)$), with additional axioms to make time points a total order. Due to lack of space, we elide the rules for establishing this judgment.

Assertions manifest in the judgment $\Sigma; \Gamma; \Delta \vdash \mu$ are established by an analysis of the program in μ through rules described in Section 4.1. Additionally, there are inference rules to combine reasoning in the temporal logic with reasoning about assertions. For instance, if $[e]\langle u_b, u_e, i, x\rangle\varphi$ and $\varphi \supset \varphi'$, then one of the rules of inference allows deduction of $[e]\langle u_b, u_e, i, x\rangle\varphi'$. Such rules are common in program logics and we elide them also.

$$\frac{\Sigma;\Gamma;\Delta \vdash [a]\langle u_b, u_e, i, x\rangle\varphi}{\Sigma;\Gamma;\Delta \vdash [\texttt{act}\ a]\langle u_b, u_e, i, x\rangle\varphi}\text{PA} \qquad \frac{\Sigma;\Gamma;\Delta \vdash []\langle u_b, u_e, i\rangle\varphi}{\Sigma;\Gamma;\Delta \vdash \{\texttt{act}\ a\}\langle u_b, u_e, i\rangle\varphi}\text{IA}$$

$$\frac{\Sigma;\Gamma;\Delta \vdash []\langle u_b, u_m, i\rangle\varphi_1 \qquad \Sigma;\Gamma;\Delta \vdash [e_1]\langle u_m, u'_m, i, y\rangle\varphi_2 \qquad \Sigma, y;\Gamma;\Delta \vdash [e_2]\langle u'_m, u_e, i, x\rangle\varphi_3}{\Sigma;\Gamma;\Delta \vdash [\texttt{let}(e_1, y.e_2)]\langle u_b, u_e, i, x\rangle\exists y.\exists u_m.\exists u'_m.((u_b < u_m < u'_m < u_e) \wedge \varphi_1 \wedge \varphi_2 \wedge \varphi_3)}\text{PL}$$

$$\frac{\begin{array}{c}\Sigma;\Gamma;\Delta \vdash []\langle u_b, u_e, i\rangle\varphi \\ \Sigma;\Gamma;\Delta \vdash []\langle u_b, u_m, i\rangle\psi_1 \qquad \Sigma;\Gamma;\Delta \vdash \{e_1\}\langle u_m, u_e, i\rangle\psi_2 \qquad \Sigma, u_b, u_m, u_e, i;\Gamma, u_b < u_m \le u_e, \psi_1, \psi_2 \vdash \varphi \\ \Sigma;\Gamma;\Delta \vdash []\langle u_b, u_m, i\rangle\psi_3 \qquad \Sigma;\Gamma;\Delta \vdash [e_1]\langle u_m, u'_m, i, y\rangle\psi_4 \\ \Sigma, y;\Gamma;\Delta \vdash \{e_2\}\langle u'_m, u_e, i\rangle\psi_5 \qquad \Sigma, u_b, u_m, u'_m, u_e, i, y;\Gamma, u_b < u_m < u'_m \le u_e, \psi_3, \psi_4, \psi_5 \vdash \varphi\end{array}}{\Sigma;\Gamma;\Delta \vdash \{\texttt{let}(e_1, y.e_2)\}\langle u_b, u_e, i\rangle\varphi}\text{IL}$$

$$\frac{f(z) \triangleq e \qquad \Sigma, y;\Gamma;\Delta, [f]\langle y, u_b, u_e, i, x\rangle\varphi \vdash [e\{y/z\}]\langle u_b, u_e, i, x\rangle\varphi}{\Sigma;\Gamma;\Delta \vdash [f]\langle y, u_b, u_e, i, x\rangle\varphi}\text{PF}$$

$$\frac{f(z) \triangleq e \qquad \Sigma, y;\Gamma;\Delta, \{f\}\langle y, u_b, u_e, i\rangle\varphi \vdash \{e\{y/z\}\}\langle u_b, u_e, i\rangle\varphi}{\Sigma;\Gamma;\Delta \vdash \{f\}\langle y, u_b, u_e, i\rangle\varphi}\text{IF}$$

Fig. 3. Selected modular rules for establishing program specifications

4.1 Reasoning About Specifications of Programs

Representative rules for establishing program specifications are shown in Figure 3. As mentioned in Section 3.3, the rules of our proof system rely on the abstract judgments $[a]\langle u_b, u_e, i, x\rangle\varphi$ and $[]\langle u_b, u_e, i\rangle\varphi$ (e.g., rule (PA)). The rules are modular: specifications of a program are established by combining specifications of sub-programs. For instance, we may justify the rule (PL) as follows. In the conclusion of the rule we wish to establish a partial correctness assertion of the expression $\texttt{let}(e_1, y.e_2)$. If this expression is active in thread i at time u_b and returns value x at time u_e, then through an analysis of the operational semantics it follows that at some time u_m after u_b, e_1 must have become active, then at a later time u'_m, e_1 would have returned some value y to e_2, which would have become active, and finally at time u_e, e_2 would have returned x. So if $[]\langle u_b, u_m, i\rangle\varphi_1$, $[e_1]\langle u_m, u'_m, i, y\rangle\varphi_2$, and $[e_2]\langle u'_m, u_e, i, x\rangle\varphi_3$ all hold (as in the premises of the rule), then $\exists y.\exists u_m.\exists u'_m.((u_b < u_m < u'_m < u_e) \wedge \varphi_1 \wedge \varphi_2 \wedge \varphi_3)$ must hold. Observe that it is necessary to existentially quantify the variables y, u_m, and u'_m in the conclusion because during reasoning, we cannot determine their exact values. This justifies the conclusion of the rule. Other rules for establishing partial correctness assertions can be justified similarly.

Rules for establishing invariance assertions are more involved, but are also modular. We illustrate their justification through the rule (IL). In the conclusion of the rule we wish to establish an invariant that holds while $\texttt{let}(e_1, y.e_2)$ executes. If this expression is active in thread i at time u_b but does not return until time u_e, then there are only three possibilities: (a) e_1 does not start executing until time u_e, (b) e_1 starts executing at some time u_m, but does not return until time u_e, or (c) e_1 starts executing at time u_m, returns at time u'_m, e_2 starts executing at time u'_m, but does not return until time u_e. If we can show that φ holds in each of these three cases, then φ is in invariant of $\texttt{let}(e_1, y.e_2)$. The premises of the rule account for exactly these three cases: the first premise accounts for case (a), premises 2–4

account for case (b), and the remaining premises account for case (c).

Rules (PF) and (IF) for proving partial correctness assertions and invariants of functions check the corresponding specification on the bodies of the respective functions. In order to account for the possibility of recursion, we also assume the function's specification when we check the body of the function by adding it to the context Δ in the premises. It is not obvious that this approach is sound and accounting for it complicates our proof of soundness (see Section 5).

Example 3. The invariant in Example 2 can be proved using the rules presented in this section, related rules for other types of program expressions, and some straightforward axioms for relevant actions. We list two such axioms below, leaving the rest to the full version of the paper. These axioms mean that the receive action `recv` and administrative reductions do not insert anything into the access control matrix. Soundness of such axioms is easy to prove, as has been demonstrated in prior work [13,14].

$$\vdash [\texttt{recv}]\langle u_b, u_e, i, x\rangle\ \forall u, k, f, p.\ ((u_b < u \leq u_e) \supset (\neg(\texttt{Insert}(i, (f, k, p)) \ @\ u)))$$

$$\vdash [][\langle u_b, u_e, i\rangle\ \forall u, k, f, p.\ ((u_b < u \leq u_e) \supset (\neg(\texttt{Insert}(i, (f, k, p)) \ @\ u)))$$

4.2 Reasoning About Trusted Threads

Next, we present the rule used to prove properties of trusted threads from knowledge of their programs. In the logic, we say that $\texttt{HonestThread}(I, e)$ if thread I executes program expression e only. Let $\texttt{Start}(I)\ @\ u$ hold if at time u, thread I is ready to execute, but has not performed any reduction. The following rule, based on the Honesty rule in prior work on Protocol Composition Logic [13], allows us to prove a property of thread I from an invariant of e if $\texttt{HonestThread}(I, e)$.

$$\frac{\Sigma; \Gamma; \cdot \vdash \{e\}\langle u_b, u_e, i\rangle\varphi(u_b, u_e, i) \qquad \Sigma; \Gamma \vdash \texttt{HonestThread}(I, e) \qquad \Sigma; \Gamma \vdash \texttt{Start}(I)\ @\ u}{\Sigma; \Gamma \vdash \forall u'.\ (u' > u) \supset \varphi(u, u', I)}\text{HONTH}$$

The justification for this rule is the following: since $\texttt{HonestThread}(I, e)$ and $\texttt{Start}(I)\ @\ u$, it must be the case that e is the active expression in I at time u. Further, since e is the top-level program of I, it can never return. Hence, by the definition of $\{e\}\langle u_b, u_e, i\rangle\varphi(u_b, u_e, i)$, $\varphi(u, u', I)$ must hold for any $u' > u$. We do not stipulate rules for proving either $\texttt{HonestThread}(I, e)$ or $\texttt{Start}(I)\ @\ u$ since they codify system-specific assumptions. Instead such formulas must be explicit hypotheses in Γ, as illustrated below.

Example 4. We demonstrate an application of the (HONTH) rule on the invariant of Example 2. Suppose that the teacher's program is executing in thread i_0 since the beginning of time ($-\infty$). Since we know that the teacher's program is

`teacher_body`, and we assume that this is the only program the teacher executes, we can assume that:

(A) $\texttt{HonestThread}(i_0, \texttt{teacher\_body})$

(B) $\forall i.\ (\texttt{Owner}(i, \mathsf{Teacher}) \supset (i = i_0))$

(C) $\texttt{Start}(i_0)\ @\ -\infty$

Applying the rule (HONTH) to the invariant from Example 2 and (A), (C) above, we can conclude that

$$\forall u'.\ (u' > -\infty) \supset (\forall u, f, k.\ ((-\infty < u \leq u') \supset (\neg\texttt{Insert}(i_0, (f, k, \texttt{ad}))\ @\ u)))$$

Simplifying and combining with (B), we get

$$\forall i, u.\ \texttt{Owner}(i, \mathsf{Teacher}) \supset \neg(\texttt{Insert}(i, (f, k, \texttt{ad}))\ @\ u)$$

In simple words, the statement above means that the teacher never inserts the administrate permission into the access control matrix, which is property (1) of the proof outline at the end of Section 3.2.

4.3 Reasoning About Interface-Confined Untrusted Threads

As opposed to trusted threads, whose security properties may be established by analysis of their programs, the programs of untrusted or adversarial threads are not known, so proving their security properties may seem impossible. Yet, in practice, security of systems often relies on confinement of behavior of untrusted threads. For instance, property (2) in the proof outline of Section 3.2, which states that a thread must have administrate permission to add a permission, holds of all threads including those that are untrusted, e.g., threads run by students and adversaries. The property holds because the only interface that allows modification to the access control matrix (`permadd`) checks for the administrate permission. In general, relevant properties of untrusted threads can often be proved by an analysis of the interfaces they have access to, even if we do not know the exact code the threads execute. In this section, we develop reasoning principles to perform such reasoning in the proof system of our logic. We define an interface, denoted $\mathcal{F}, \mathcal{G}$, as a set of functions. In general, an untrusted thread that is confined to $\mathcal{F}$ may construct a new set of functions $\mathcal{G}$ that call functions of $\mathcal{F}$ and themselves and combine calls to functions of $\mathcal{F}$ and $\mathcal{G}$ in any way it chooses. To formally represent such an adversary, we need a few definitions.

Definition 4.1 ($\mathcal{F}$-confined expressions) *Given an interface $\mathcal{F}$, we call an expression e $\mathcal{F}$-confined if the following hold: (a) All occurrences of* `call` *in e have the form* $\texttt{call}(f, t)$*, where $f \in \mathcal{F}$, and (b)* `act` *does not occur in e.*

Definition 4.2 ($\mathcal{F}$-limited functions) *Given an interface $\mathcal{F}$, we call a set of functions $\mathcal{G} = \{g_k \mid g_k(y) \stackrel{\triangle}{=} e_k\}$ $\mathcal{F}$-limited if the body e_k of each function is $(\mathcal{F} \cup \mathcal{G})$-confined.*

Definition 4.3 ($\mathcal{F}$-confined thread) *A (untrusted) thread I is said to be $\mathcal{F}$-confined if I executes a program e and there is a $\mathcal{F}$-limited interface $\mathcal{G}$ such that e is $(\mathcal{F} \cup \mathcal{G})$-confined. The predicate* $\texttt{Confined}(I, \mathcal{F})$ *holds iff I is $\mathcal{F}$-confined.* [2]

Definition 4.4 (Compositional formula) *A formula $\varphi(u_b, u_e, i)$, possibly containing the free variables u_b, u_e, i, is called compositional if* $\forall u_b, u_m, u_e, i.\ ((u_b < u_m \leq u_e) \wedge \varphi(u_b, u_m, i) \wedge \varphi(u_m, u_e, i)) \supset \varphi(u_b, u_e, i)$.

Roughly, a formula $\varphi(u_b, u_e, i)$ describing some property over an interval $(u_b, u_e]$ is compositional if whenever the formula holds on two adjoining intervals, it also holds on the union of the intervals. In general, if $\varphi(u_b, u_e, i)$ encodes the fact that a safety property holds throughout the interval $(u_b, u_e]$, then $\varphi(u_b, u_e, i)$ will be compositional.

We codify our reasoning principles for untrusted, interface-confined threads in the following rule:

$$\frac{\begin{array}{c}(\varphi(u_b, u_e, i)\ \text{compositional}) \\ \cdot; \Gamma; \cdot \vdash [] \langle u_b, u_e, i \rangle \varphi(u_b, u_e, i) \qquad \forall f \in \mathcal{F}.\ (\cdot; \Gamma; \cdot \vdash \{f\} \langle y, u_b, u_e, i \rangle \varphi(u_b, u_e, i)) \\ \forall f \in \mathcal{F}.\ (\cdot; \Gamma; \cdot \vdash [f] \langle y, u_b, u_e, i, x \rangle \varphi(u_b, u_e, i)) \qquad \Sigma; \Gamma \vdash \texttt{Confined}(I, \mathcal{F})\end{array}}{\Sigma; \Gamma, \Gamma' \vdash \forall u_e.\ \varphi(-\infty, u_e, I)}\ \text{RES}$$

The informal justification for the rule (RES) is that, owing to its confinement to $\mathcal{F}$, the reduction of I up to any time point u_e can be split into calls to functions in $\mathcal{F}$ interspersed with administrative reductions of the adversary's choosing. Since φ is a partial correctness assertion of all functions in $\mathcal{F}$ (fourth premise) and administrative reductions (second premise), as well an invariant of all functions in $\mathcal{F}$ (third premise), it must hold over all these splits. Therefore, due to the compositionality of φ (first premise), $\varphi(-\infty, u_e, I)$ must hold. The formal justification of this rule is a non-trivial part of the soundness theorem (Section 5) because we must consider all $\mathcal{F}$-confined programs that the thread I may execute.

Example 5. We sketch a proof of property (2) from the outline at the end of Section 3.2. The property states that if a thread can insert a permission for file f into the access control matrix then the thread must have the administrate permission on f, or, formally,

$$\begin{array}{l}\forall i, f, k, k', p, u, s.\ ((\texttt{Insert}(i, (f, k', p))\ @\ u) \wedge \texttt{Owner}(i, k)) \\ \qquad \supset \exists u'.\ (u' < u \wedge (\texttt{isACM}(s)\ @\ u') \wedge ((f, k, \texttt{ad}) \in s))\end{array}$$

[2] The restriction that the untrusted thread may not execute any actions directly may seem to limit expressiveness of our model because we may want to allow the untrusted thread access to some actions. However, this is not really a limitation because we may give the thread access to interfaces that execute the allowed actions immediately. For instance, to allow an adversary access to the `send` action, we may give it the interface $f(x) \triangleq \texttt{send}\ x$.

where $\texttt{isACM}(s)$ means that the (current) access control matrix is the set of tuples s. We prove this property using the (RES) rule. Define

$$\begin{aligned}\varphi(u_b, u_e, i) = \forall f, k, k', p, u, s.\\ ((u_b < u \leq u_e) \wedge (\texttt{Insert}(i, (f, k', p)) \mathbin{@} u) \wedge \texttt{Owner}(i, k))\\ \supset \exists u'.\ (u' < u \wedge (\texttt{isACM}(s) \mathbin{@} u') \wedge ((f, k, \texttt{ad}) \in s))\end{aligned}$$

Then, using the rules presented in Section 4.1, we can prove the following for the interface $\mathcal{F}$ defined in Figure 2:

$$\forall g \in \mathcal{F}.\ (\{g\}\langle y, u_b, u_e, i\rangle\varphi(u_b, u_e, i))$$
$$\forall g \in \mathcal{F}.\ ([g]\langle y, u_b, u_e, i, x\rangle\varphi(u_b, u_e, i))$$
$$[]\langle u_b, u_e, i\rangle\varphi(u_b, u_e, i)$$

Further, since all threads are assumed to be confined to the interface $\mathcal{F}$, we can assume that $\forall i.\ \texttt{Confined}(i, \mathcal{F})$. Finally, it can be easily checked that $\varphi(u_b, u_e, i)$ is compositional. Hence, by rule (RES) we conclude that $\forall i. \forall u_e.\ \varphi(-\infty, u_e, i)$. This implies the formula at the beginning of this example in a straightforward manner.

5 Semantics and Soundness Theorem

We formally define semantics of temporal formulas φ and assertions μ with respect to traces and show that our proof rules are sound, i.e., any formula or assertion proved using the rules is valid in the semantics. This provides foundational justification for the reasoning principles of Section 4.

Semantics. Since our programming model and the logic of programs are parametric in the syntax of terms and predicates, we assume that interpretations of these entities are given. Let $[\![t]\!]$ denote the semantic interpretation of the term t in some domain and let $\doteq$ denote equality in the domain. For interpreting atomic formulas, we assume the existence of a Boolean valued function $V(\mathcal{T}, u, p)$ ($\mathcal{T}$ is a trace, u is a ground time point, and p is a ground atomic formula) such that $[\![t]\!] \doteq [\![t']\!]$ implies $V(\mathcal{T}, u, p\{t/x\}) = V(\mathcal{T}, u, p\{t'/x\})$. Given these assumptions, we may define the semantics $\mathcal{T} \models \varphi$ of ground temporal formulas φ in a standard manner. For example,

- $\mathcal{T} \models p \mathbin{@} u$ iff $V(\mathcal{T}, u, p)$.
- $\mathcal{T} \models \varphi \wedge \psi$ iff $\mathcal{T} \models \varphi$ and $\mathcal{T} \models \psi$.

In order to define semantics of assertions, we need a notion of the suffix of a trace, also called a subtrace.

Definition 5.1 (Subtraces) *Let $\mathcal{T}$ be the trace*

$$\xrightarrow{u_0} \mathcal{C}_0 \xrightarrow{u_1} \mathcal{C}_1 \ldots \xrightarrow{u_n} \mathcal{C}_n$$

For any $k \geq 0$, we define the truncation of $\mathcal{T}$ to k, written $\texttt{trunc}(\mathcal{T}, k)$ as the trace which contains only the last $k+1$ configurations of $\mathcal{T}$. If $k > n$ then $\texttt{trunc}(\mathcal{T}, k) = \mathcal{T}$.

Semantics of ground assertions μ are represented through the judgment $\mathcal{T}, n \models \mu$, which roughly means that the assertion μ holds in the subtrace $\texttt{trunc}(\mathcal{T}, n)$. The additional information n is needed to prove soundness for recursive functions. One representative clause of the definition of $\mathcal{T}, n \models \mu$ is shown below; others are described in the full version of this paper.

- $\mathcal{T}, n \models \{e\}\langle u_b, u_e, i\rangle\varphi$ holds iff $\mathcal{T} \models \varphi\{u'_b/u_b\}\{u'_e/u_e\}\{I/i\}$ whenever the following pattern matches the subtrace $\texttt{trunc}(\mathcal{T}, n)$ for some $u' < u'_b$, there is no reduction in thread I in the interval $(u', u'_b]$, and the stack of I has suffix K throughout the interval $(u'_b, u'_e]$.

$$\cdots \xrightarrow{u'} \sigma \rhd I\,;K\,;e$$

Finally, we define semantics of hypothetical judgments of the proof system.

- $\mathcal{T} \models (\Sigma; \Gamma \vdash \varphi)$ if for every grounding substitution θ with domain Σ, $\mathcal{T} \models \Gamma\theta$ implies $\mathcal{T} \models \varphi\theta$.
- $\mathcal{T} \models (\Sigma; \Gamma; \Delta \vdash \mu)$ if for every grounding substitution θ with domain Σ and every n, $\mathcal{T} \models \Gamma\theta$ and $\mathcal{T}, n \models \Delta\theta$ imply $\mathcal{T}, n \models \mu\theta$.

Soundness. We show that any hypothetical judgment established using the proof system of Section 4 is semantically valid, if the axioms chosen to reason about the assertions $[a]\langle u_b, u_e, i, x\rangle\varphi$ and $[]\langle u_b, u_e, i\rangle$ are valid in the semantics. As a result, any instance of our reasoning principles is sound if we choose sound axioms for actions and administrative reductions.

Theorem 5.2 (Soundness) *Suppose that each assumed axiom (e.g., about the assertions $[a]\langle u_b, u_e, i, x\rangle\varphi$ and $[]\langle u_b, u_e, i\rangle\varphi$) is sound. Then for every $\mathcal{T}$,*

(i) $\Sigma; \Gamma \vdash \varphi$ *implies* $\mathcal{T} \models (\Sigma; \Gamma \vdash \varphi)$.

(ii) $\Sigma; \Gamma; \Delta \vdash \mu$ *implies* $\mathcal{T} \models (\Sigma; \Gamma; \Delta \vdash \mu)$.

The proof of soundness proceeds by a lexicographic induction, first on the maximum number of (RES) rules in any path in the given derivation, and then on the depth of the derivation. A simpler, more obvious induction on the depth of the derivation does not work because in the (RES) rule the proof that the program e being executed by the thread I satisfies invariant φ may be arbitrarily deeper than the proofs of the premises. Another technical difficulty arises due to the possibility of recursive functions: for the rules (PF) and (IF) of Figure 3, we must subinduct on the number n in the definition of $\mathcal{T}, n \models \mu$.

6 Rely-Guarantee Reasoning

Often, a security property relies on specific behavior of threads that can be ascertained only if the security property itself holds in the past. For example, consider property (3) from the outline at the end of Section 3.2: only the teacher ever obtains administrate permissions. Reasoning that this property holds at any point of time relies on the property having been true at all points of time in the past. Similarly, the analysis of secrecy of keys in security protocols often relies both on the keys having remained secret in the past as illustrated in prior work [33].

The rely-guarantee method is a general technique for proving such properties for concurrently executing threads [27,21,15]. Summarily, suppose φ is a property of state. The rely-guarantee method envisages that in order to show that φ holds in all states of the system's execution, it suffices to prove the following three properties:

(A) φ holds initially.

(B) There is a class of properties $\psi(i)$, indexed by threads i, such that for any action that i may perform, if φ holds in the state preceding the action, then $\psi(i)$ holds immediately after i executes the action.

(C) If φ holds in a state and $\psi(i)$ holds in the next state for all i, then φ holds in the next state.

Here, we show how, for a wide class of properties, the rely-guarantee technique is a special case of the reasoning principles presented in Section 4. Suppose $\varphi(u)$ is a property that we wish to establish for all time points u. Assume that a predicate $\iota(i)$ identifies threads of interest and there is a thread-specific property $\psi(u, i)$ such that the following analogues of the properties (A)–(C) hold:

(A') $\varphi(-\infty)$

(B') $\forall i, u.\ (\iota(i) \wedge \forall u' < u.\ \varphi(u')) \supset \psi(u, i)$

(C') $(\varphi(u_1) \wedge \neg\varphi(u_2) \wedge (u_1 < u_2)) \supset$

$$\exists i, u_3.\ (u_1 < u_3 \leq u_2) \wedge \iota(i) \wedge \neg\psi(u_3, i) \wedge \forall u_4 \in (u_1, u_3).\ \varphi(u_4)$$

Then, we can prove using elementary logical reasoning that $\forall u.\varphi(u)$ (see the next Theorem). Roughly, condition (B') is analogous to property (B) and it may be established through invariants. For instance, if $\iota(i) = i \in \mathcal{I}$, where $\mathcal{I}$ is a set of trusted threads, then by rule (HONTH), condition (B') holds if the following assertion holds for all programs e that threads in $\mathcal{I}$ may execute.

$$\{e\}\langle u_b, u_e, i\rangle\ \forall u \in (u_b, u_e].\ (\forall u' < u.\ \varphi(u')) \supset \psi(u, i)$$

If, on the other hand, $\iota(i) = \top$ (i.e., all threads are of interest), then we may prove (B') using the (RES) rule. Condition (C') means that if there is a violation of φ at time u_2 but this was not the case at time u_1 $(u_1 < u_2)$ then there must be a first violation at some time u_3 that is caused due to a violation of the thread specific property by some thread satisfying ι. This is stronger than property (C), but holds

for properties φ that depend solely on state. In general, a rigorous proof of a condition like (C') requires induction over traces. Since we do not have a principle for induction on trace in our formalism, we must, in some cases, derive (C') from an axiom, which must then be proved sound in the semantic model by induction on traces.

Theorem 6.1 (Rely-guarantee) *Conditions (A')–(C') as above imply* $\forall u.\ \varphi(u)$ *in the proof system of Section 4.*

Proof. By a straightforward analysis in the temporal logic. □

Example 6. We outline briefly how property (3) mentioned at the end of Section 3.2 can be proved using the rely-guarantee method described above. Define the predicate $\texttt{HasOnly}(\mathcal{K}, f, p, u)$ to mean that only principals in set $\mathcal{K}$ have permission p on file f at time u:

$$\texttt{HasOnly}(\mathcal{K}, f, p, u) \ = \ \forall k, s.\ ((\texttt{isACM}(s) \mathbin{@} u) \wedge ((f, k, p) \in s)) \supset (k \in \mathcal{K})$$

Then, we can prove by an induction on traces that the following axiom (PERM) is sound:

$$\begin{aligned}
&\forall \mathcal{K}, f, u_1, u_2, p.\\
&\quad (\texttt{HasOnly}(\mathcal{K}, f, p, u_1) \wedge \neg\texttt{HasOnly}(\mathcal{K}, f, p, u_2) \wedge (u_1 < u_2))\\
&\qquad \supset (\exists i, k', u_3.\ (u_1 < u_3 \leq u_2) \wedge (k' \notin \mathcal{K}) \wedge (\texttt{Insert}(i, (f, k', p)) \mathbin{@} u_3) \wedge\\
&\qquad\qquad \forall u_4.\ (u_1 < u_4 < u_3) \supset \texttt{HasOnly}(\mathcal{K}, f, p, u_4))
\end{aligned}$$

Intuitively, the axiom states that if at time u_1 only principals in $\mathcal{K}$ have permission p on file f and at later time u_2 this is not the case, then there must be a first time u_3 when some thread i inserted permission p on f for a principal $k' \notin \mathcal{K}$.

The property (3) we wish to prove can be formalized as $\forall u.\varphi(u)$ where

$$\varphi(u) \ = \ \forall n.\ \texttt{HasOnly}(\{\mathsf{Teacher}\}, f[n], \mathsf{ad}, u)$$

Following the rely-guarantee method described above, define

$$\begin{aligned}
\iota(i) \quad &= \quad \top\\
\psi(u, i) \quad &= \quad \forall n, k'.\ (k' \neq \mathsf{Teacher}) \supset \neg(\texttt{Insert}(i, (f[n], k', \mathsf{ad})) \mathbin{@} u)
\end{aligned}$$

By Theorem 6.1, if we can prove (A')–(C') for the φ, ψ, and ι defined above, then there is a proof of the required property $\forall u.\ \varphi(u)$. We analyze each of these conditions. (A') is the formula $\varphi(-\infty) = \forall n.\ \texttt{HasOnly}(\{\mathsf{Teacher}\}, f[n], \mathsf{ad}, -\infty)$, which holds because we assumed that initially only Teacher has administrate permissions. (Technically, (A') is an axiom in our example.) After substitution of φ, ψ, and ι, (C') is easily seen to be equivalent to axiom (PERM) written earlier

and, therefore, has an immediate proof. The statement of (B') is:

$$\forall i, u.\ (\forall u'.\ ((u' < u) \supset \forall n.\ \texttt{HasOnly}(\{\mathsf{Teacher}\}, f[n], \mathsf{ad}, u'))) \\ \supset \forall n, k'.\ (k' \neq \mathsf{Teacher}) \supset \neg(\texttt{Insert}(i, (f[n], k', \mathsf{ad})) \ @\ u)$$

This follows from the properties proved in Examples 4 and 5.

7 Future Work

This paper makes significant progress towards developing a systematic foundation for compositional system security. We plan to extend this work in several directions. So far, we have considered reasoning principles for first-order programs where code cannot be passed as arguments or returned from expressions. However, many systems rely on passing code either as data or through pointers. To model and to establish security properties of such applications, we propose to extend the formalism with higher-order constructs and develop associated compositional reasoning principles. While this paper has focused on the technical foundations of the theory, we plan to apply this framework to develop a systematic basis for web security, to formalize attacker models for web browsers proposed in the literature [4] and develop new ones, and to build an understanding of relevant security policies, end-to-end security properties, attacks in the wild, and ways to defend and prove web applications secure against these attacks.

Acknowledgement

This work was partially supported by the U.S. Army Research Office contract on Perpetually Available and Secure Information Systems (DAAD19-02-1-0389) to CMUs CyLab, the NSF Science and Technology Center TRUST, and the NSF CyberTrust grant "Realizing Verifiable Security Properties on Untrusted Computing Platforms". Jason Franklin is supported in part by an NSF Graduate Research Fellowship.

References

[1] Alpern, B. and F. B. Schneider, *Recognizing safety and liveness*, Distributed Computing **2** (1987), pp. 117–126.

[2] Asokan, N., V. Niemi and K. Nyberg, *Man-in-the-middle in tunnelled authentication protocols*, in: *Security Protocols Workshop*, 2003, pp. 28–41.

[3] Barth, A., C. Jackson and J. C. Mitchell, *Robust defenses for cross-site request forgery*, in: *Proceedings of the 15th ACM Conference on Computer and Communications Security (CCS 2008)*, 2008, pp. 75–88.

[4] Barth, A., C. Jackson and J. C. Mitchell, *Securing frame communication in browsers*, in: *Proceedings of the 17th USENIX Security Symposium*, 2008, pp. 17–30.

[5] Bellovin, S., *Security challenges*, in: *1st ITI Workshop on Dependability and Security*, 2004, panel: Grand challenges and open questions in trusted systems.

[6] Bhargavan, K., C. Fournet and A. D. Gordon, *Modular verification of security protocol code by typing*, in: *Proceedings of the 37th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages (POPL)*, 2010, to appear.

[7] Blackburn, P., *Representation, reasoning, and relational structures: A hybrid logic manifesto*, Logic Journal of IGPL **8** (2000), pp. 339–365.

[8] Cai, X., Y. Gui and R. Johnson, *Exploiting Unix file-system races via algorithmic complexity attacks*, in: *SP '09: Proceedings of the 2009 30th IEEE Symposium on Security and Privacy*, 2009, pp. 27–41.

[9] Canetti, R., *Universally composable security: A new paradigm for cryptographic protocols*, in: *FOCS*, 2001, pp. 136–145.

[10] Chen, S., Z. Mao, Y.-M. Wang and M. Zhang, *Pretty-bad-proxy: An overlooked adversary in browsers' HTTPS deployments*, in: *Proceedings of the 30th IEEE Symposium on Security and Privacy*, 2009, pp. 347–359.

[11] Cortier, V. and S. Delaune, *Safely composing security protocols*, Formal Methods in System Design **34** (2009), pp. 1–36.

[12] Datta, A., A. Derek, J. C. Mitchell and D. Pavlovic, *A derivation system and compositional logic for security protocols*, Journal of Computer Security **13** (2005), pp. 423–482.

[13] Datta, A., A. Derek, J. C. Mitchell and A. Roy, *Protocol Composition Logic (PCL)*, Electronic Notes in Theoretical Computer Science **172** (2007), pp. 311–358.

[14] Datta, A., J. Franklin, D. Garg and D. Kaynar, *A logic of secure systems and its application to trusted computing*, in: *Proceedings of the 30th IEEE Symposium on Security and Privacy (Oakland)*, 2009, pp. 221–236.

[15] Feng, X., R. Ferreira and Z. Shao, *On the relationship between concurrent separation logic and assume-guarantee reasoning*, in: *Programming Languages and Systems, Proceedings of the 16th European Symposium on Programming (ESOP)*, 2007, pp. 173–188.

[16] Garg, D., J. Franklin, A. Datta and D. Kaynar, *Compositional system security in the presence of interface-confined adversaries* (2010), full version. Online at http://www.cs.cmu.edu/~dg.

[17] Guttman, J. D. and F. J. Thayer, *Protocol independence through disjoint encryption*, in: *CSFW*, 2000, pp. 24–34.

[18] He, C., M. Sundararajan, A. Datta, A. Derek and J. C. Mitchell, *A modular correctness proof of IEEE 802.11i and TLS*, in: *CCS '05: Proceedings of the 12th ACM Conference on Computer and Communications Security*, 2005, pp. 2–15.

[19] Jackson, C. and A. Barth, *Forcehttps: protecting high-security web sites from network attacks*, in: *Proceedings of the 17th International Conference on World Wide Web (WWW)*, 2008, pp. 525–534.

[20] Jackson, C., A. Barth, A. Bortz, W. Shao and D. Boneh, *Protecting browsers from DNS rebinding attacks*, 2007, pp. 421–431.

[21] Jones, C. B., *Tentative steps toward a development method for interfering programs*, ACM Transactions on Programming Languages and Systems (TOPLAS) **5** (1983), pp. 596–619.

[22] Kuhlman, D., R. Moriarty, T. Braskich, S. Emeott and M. Tripunitara, *A correctness proof of a mesh security architecture*, in: *Proceedings of the 21st IEEE Computer Security Foundations Symposium (CSF)*, 2008, pp. 315–330.

[23] Mantel, H., *On the composition of secure systems*, in: *SP '02: Proceedings of the 2002 IEEE Symposium on Security and Privacy* (2002), pp. 88–101.

[24] McCullough, D., *A hookup theorem for multilevel security*, IEEE Transactions on Software Engineering **16** (1990), pp. 563–568.

[25] McLean, J., *Security models and information flow*, in: *IEEE Symposium on Security and Privacy*, 1990, pp. 180–189.

[26] Meadows, C. and D. Pavlovic, *Deriving, attacking and defending the gdoi protocol*, in: *Proceedings of the 9th European Symposium on Research in Computer Security (ESORICS)*, 2004, pp. 53–72.

[27] Misra, J. and K. M. Chandy, *Proofs of networks of processes*, IEEE Transactions on Software Engineering **7** (1981), pp. 417–426.

[28] Mitchell, J. C., *Programming language methods in computer security*, in: *Proceedings of the 28th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*, 2001, pp. 1–3.

[29] Mitchell, J. C., V. Shmatikov and U. Stern, *Finite-state analysis of ssl 3.0*, in: *SSYM'98: Proceedings of the 7th conference on USENIX Security Symposium*, 1998, pp. 16–16.

[30] Nanevski, A., G. Morrisett and L. Birkedal, *Hoare type theory, polymorphism and separation*, Journal of Functional Programming **18** (2008), pp. 865–911.

[31] Pfitzmann, B. and M. Waidner, *A model for asynchronous reactive systems and its application to secure message transmission*, in: *IEEE Symposium on Security and Privacy*, 2001, pp. 184–.

[32] Reed, J., "A Hybrid Logical Framework," Ph.D. thesis, Carnegie Mellon University (2009).

[33] Roy, A., A. Datta, A. Derek, J. C. Mitchell and S. Jean-Pierre, *Secrecy analysis in protocol composition logic*, in: *Formal Logical Methods for System Security and Correctness*, IOS Press, 2008 .

[34] Tsafrir, D., T. Hertz, D. Wagner and D. Da Silva, *Portably solving file tocttou races with hardness amplification*, in: *FAST'08: Proceedings of the 6th USENIX Conference on File and Storage Technologies*, 2008, pp. 1–18.

[35] Wing, J. M., *A call to action: Look beyond the horizon*, IEEE Security & Privacy **1** (2003), pp. 62–67.